A Type System Theory for Higher-Order Intensional Logic Support for Variable Bindings in Hybrid Intensional-Imperative Programs in GIPSY

Serguei A. Mokhov
Concordia University, Montreal, Canada
mokhov@cse.concordia.ca

Joey Paquet
Concordia University, Montreal, Canada
paquet@cse.concordia.ca

Abstract

We describe a type system for a platform called the General Intensional Programming System (GIPSY), designed to support intensional programming languages built upon intensional logic and their imperative counter-parts for the intensional execution model. In GIPSY, the type system glues the static and dynamic typing between intensional and imperative languages in its compiler and run-time environments to support the intensional evaluation of expressions written in various dialects of the intensional programming language Lucid. The intensionality makes expressions to explicitly take into the account a multidimensional context of evaluation with the context being a first-class value that serves a number of applications that need the notion of context to proceed. We describe and discuss the properties of such a type system and the related type theory as well as particularities of the semantics, design and implementation of the GIPSY type system.

Keywords: Intensional Programming, Imperative Programming, Type System, Type Theory, General Intensional Programming System (GIPSY), Context

1 Introduction

The GIPSY is an ongoing effort for the development of a flexible and adaptable multi-lingual programming language development framework aimed at the investigation on the Lucid family of intensional programming languages [2, 2, 22, 1, 12, 11, 17, 18]. Using this platform, programs written in many flavors of Lucid can be compiled and executed. The framework approach adopted is aimed at providing the possibility of easily developing compiler components for other languages of intensional nature, and to execute them on a language-independent run-time system. Lucid being a functional "data-flow" language, its programs can be executed in a distributed processing environment. However, the standard Lucid algebra (i.e. types and operators) is extremely fine-grained and can hardly benefit from parallel evaluation of their operands. Adding granularity to the data elements manipulated by Lucid inevitably comes through the addition of coarser-grained data types and their corresponding operators. Lucid semantics being defined as typeless, a solution to this problem consists in adding a hybrid counterpart to Lucid to allow an external language to define an algebra of coarser-grained types and operators. In turn, this solution raises the need for an elaborated type system to bridge the implicit types of Lucid semantics with the explicit types and operators (i.e. functions) defined in the hybrid counterpart language. This paper presents such a type system that is used in the GIPSY at compile time for static type checking, as well as at run-time for dynamic type checking.

After our problem statement and short presentation of our proposed solution, this section presents a brief introduction to the GIPSY project [14, 6, 13, 21, 10, 25, 16], and the beginnings of the GIPSY Type System [10] in the GIPSY to support hybrid and object-oriented intensional programming [9, 24], and Lucx's context type extension known as context calculus [20, 15, 19] for contexts to act as first-class values. Then, we present the complete GIPSY type system as used by the compiler (General Intensional Programming Compiler – GIPC) and the run-time system (General Eduction Engine – GEE) when producing and executing a binary GIPSY program (called General Eduction Engine Resources – GEER) respectively.

1.1 Problem Statement and Proposed Solution

Problem Statement Data types are implicit in Lucid (as well as in its dialects or many functional languages). As such, the type declarations never appear in the Lucid programs at the syntactical level. The data type of a value is inferred from the result of evaluation of an expression. In most imperative languages, like Java, C++ and the like, the types are explicit and the programmers must *declare* the types of the variables, function parameters and return values before they are used in evaluation. In the GIPSY, we want to allow any Lucid dialect to be able to uniformly invoke functions/methods written in imperative languages and the other way around and perform semantic analysis in the form of type assignment and checking statically at compile time or dynamically at run time, perform any type conversion if needed, and evaluate such hybrid expressions. At the same time, we need to allow a programmer to specify, or declare, the types of variables, parameters, and return values for both intensional and imperative functions as a binding contract between inter-language invocations despite the fact that Lucid is not explicitly typed. Thus, we need a general type system, well designed and implemented to support such scenarios.

Proposed Solution The uniqueness of the type system presented here is that it is above a specific programming language model of either the Lucid family of languages or imperative languages. It is designed to bridge programming language paradigms, the two most general of them would be the intensional and imperative paradigms. GIPSY has a framework designed to support a common run-time environment and co-existence of the intensional and imperative languages. Thus, the type system is that of a generic GIPSY program that can include code segments written in a theoretically arbitrary number of intensional and imperative dialects supported by the system vs. being a type system for a specific language. What follows is the details of the proposed solution and the specification of the type system.

1.2 Introduction to the GIPSY Type System

The introduction of JLucid, Objective Lucid, and the General Imperative Compiler Framework (GICF) [10, 9, 8, 4] prompted the development of the GIPSY Type System as implicitly understood by the Lucid language and its incarnation within the GIPSY to handle types in a more general manner as a glue between the imperative and intensional languages within the system. Further evolution of different Lucid variants such as Lucx introducing contexts as first-class values and JOOIP (Java Object-Oriented Intensional Programming) highlighted the need of the further development of the type system to accommodate the more general properties of the intensional and hybrid languages.

1.2.1 Matching Lucid and Java Data Types

Here we present a case of interaction between Lucid and Java. Allowing Lucid to call Java methods brings a set of issues related to the data types, especially when it comes to type checks between Lucid and Java parts of a hybrid program. This is pertinent when Lucid variables or expressions are used as parameters to Java methods and when a Java method returns a result to be assigned to a Lucid variable or used in an intensional expression. The sets of types in both cases are not exactly the same. The basic set of Lucid data types as defined by Grogono [3] is `int`, `bool`, `double`, `string`, and `dimension`. Lucid's `int` is of the same size as Java's `long`. GIPSY and Java `double`, `boolean`, and `String` are equivalent. Lucid `string` and Java `String` are simply mapped internally through `StringBuffer`; thus, one can think of the Lucid `string` as a reference when evaluated in the intensional program. Based on this fact, the lengths of a Lucid `string` and Java `String` are the same. Java `String` is also an object in Java; however, at this point, a Lucid program has no direct access to any `String`'s properties (though internally we do and we may expose it later to the programmers). We also distinguish the `float` data

Table 1: Matching data types between Lucid and Java.

Return Types of Java Methods	Types of Lucid Expressions	Internal GIPSY Types
`int`, `byte`, `long`	`int`, `dimension`	`GIPSYInteger`
`float`	`float`	`GIPSYFloat`
`double`	`double`	`GIPSYDouble`
`boolean`	`bool`	`GIPSYBoolean`
`char`	`char`	`GIPSYCharacter`
`String`	`string`, `dimension`	`GIPSYString`
`Method`	*function*	`GIPSYFunction`
`Method`	*operator*	`GIPSYOperator`
`[]`	`[]`	`GIPSYArray`
`Object`	*class*	`GIPSYObject`
`Object`	*URL*	`GIPSYEmbed`
`void`	`bool::true`	`GIPSYVoid`
Parameter Types Used in Lucid	**Corresponding Java Types**	**Internal GIPSY Types**
`string`	`String`	`GIPSYString`
`float`	`float`	`GIPSYFloat`
`double`	`double`	`GIPSYDouble`
`int`	`int`	`GIPSYInteger`
`dimension`	`int`, `String`	`Dimension`
`bool`	`boolean`	`GIPSYBoolean`
class	`Object`	`GIPSYObject`
URL	`Object`	`GIPSYEmbed`
`[]`	`[]`	`GIPSYArray`
operator	`Method`	`GIPSYOperator`
function	`Method`	`GIPSYFunction`

type for single-precision floating point operations. The `dimension` index type is said to be an integer or string (as far as its dimension tag values are concerned), but might be of other types eventually, as discussed in [20]. Therefore, we perform data type matching as presented in Table 1. Additionally, we allow `void` Java return type which will always be matched to a Boolean expression `true` in Lucid as an expression has to always evaluate to something. As for now our types mapping and restrictions are as per Table 1. This is the mapping table for the Java-to-IPL-to-Java type adapter. Such a table would exist for mapping between any imperative-to-intensional language and back, e.g. the C++-to-IPL-to-C++ type adapter.

2 Simple Theory of GIPSY Types

Our simple theory of the GIPSY types (STGT) is based on the "Simple theory of types" (STT) by Mendelson [7]. The theoretical and practical considerations are described in the sections that follows. The STT partitions the qualification domain into an ascending hierarchy of types with every individual value assigned a type. The type assignment is dynamic for the intensional dialects as the resulting type of a value in an intensional expression may not be known at compile time. The assignment of the types of constant literals is done at compile-time, however. In the hybrid system, which is mostly statically typed at the code-segment boundaries, the type assignment also occurs at compile-time. On the boundary between the intensional programming languages (IPLs) and imperative languages, prior to an imperative procedure being invoked, the type assignment to the procedure's parameters from IPL's expression is computed dynamically *and* matched against a type mapping table similar to that of Table 1. Subsequently, the when the procedure call returns back to the IPL, the type of the imperative expression is matched back through the table and assigned to the intensional expression that expects it. The same table is used when the call is made by the procedure to the IPL and back, but in the reverse order.

Further in STT, all quantified variables range over only one type making the first-order logic applicable as the underlying logic for the theory. This also means the all elements in the domain and all co-domains are of the same type. The STT states there is an atomic type that has no member elements in it, and the members of the second-high from the basic atomic type. Each type has a next higher type similarly to `succ` in Peano arithmetic and the `next` operator in Lucid. This is also consistent to describe the composite types, such as arrays and objects as they can be recursively decomposed (or "flattened", see [10]) into the primitive types to which the STT applies.

Symbolically, the STT uses primed and unprimed variables and the infix set notation of $\in$. The formulas $\Phi(x)$ rely on the fact that the unprimed variables are all of the same type. This is similar to the notion of a Lucid stream with the point-wise elements of the stream having the same type. The primed variables (x') in STT range over the next higher type. There two atomic formulas in STT of the form of identity, $x = y$, and set membership, $y \in x'$.

The STT defines the four basic axioms for the variables and the types they can range over: Identity, Extensionality, Comprehension, and Infinity. We add the Intensionality on as the fifth axiom. The variables in the definition of the Identity relationship and in the Extensionality and Comprehension axioms typically range over the elements of one of the two nearby types. In the set membership, only the unprimed variables that range over the lower type in the hierarchy can appear on the left of $\in$; conversely, the primed ones that range over higher types can only appear on the right of $\in$. The axioms are defined as:

1. **Identity**: $x = y \leftrightarrow \forall z'[x \in z' \leftrightarrow y \in z']$
2. **Extensionality**: $\forall x[x \in y' \leftrightarrow x \in z'] \rightarrow y' = z'$
3. **Comprehension**: $\exists z' \forall x[x \in z' \leftrightarrow \Phi(x)]$. This covers objects and arrays as any collection of elements here may form an object of the next, higher type. The STT states the comprehension axiom is schematic with respect to $\Phi(x)$ and the types.
4. **Infinity**: $\forall x, y[x \neq y \rightarrow [xRy \bigvee yRx]]$. There exists a non-empty binary relation R over the elements of the atomic type that is transitive, irreflexive, and strongly connected.
5. **Intensionality**: the intensional types and operators are based on the intensional logic and context calculus. These are extensively described in the works [15, 20, 23, 12] and related. This present type system accommodates the two in a common hybrid execution environment of the GIPSY. A context c is a finite subset of the relation: $c \subset \{(d,x) | d \in DIM \wedge x \in T\}$, where DIM is the set of all possible dimensions, and T is the set of all possible tags.

2.1 Types of Types

Types of types are generally referred to as *kinds*. Kinds provide categorization to the types of similar nature. While some type systems provide kinds as first class entities available to programmers, in GIPSY we do not expose this functionality in our type system at this point. However, at the implementation level there are provisions to do so that we may later decide to expose for the use of programmers. Internally, we define several broad kinds of types, presented the the sections that follow.

2.1.1 Numeric Kind

The primitive types under this category are numerical values, which are represented by `GIPSYInteger`, `GIPSYFloat`, and `GIPSYDouble`. They provide implementation of the common arithmetic operators, such as addition, multiplication and so on, as well as logical comparison operators of ordering and equality. Thus, for a numerical type T, the following common operators are provided. The resulting type of any arithmetic operator is the largest of the two operands in terms of length (the range of `double` of length say k covers the range of `int` with the length say m and if both appear as arguments to the operator, then the resulting value's type is that of without loss of information, i.e. largest in length `double`). The result of the logical comparison operators is always Boolean B regardless the length of the left-hand-side and right-hand-side numerical types.

1. $T_{max} : T_1 \geq T_2 \rightarrow T_1 \mid T_1 < T_2 \rightarrow T_2$
2. $T_{multiply} : T_k \times T_m \rightarrow T_{\max(k,m)}$
3. $T_{divide} : T_k / T_m \rightarrow T_{\max(k,m)}$
4. $T_{add} : T_k + T_m \rightarrow T_{\max(k,m)}$
5. $T_{subtract} : T_k - T_m \rightarrow T_{\max(k,m)}$
6. $T_{mod} : T_k \% T_m \rightarrow T_{\max(k,m)}$
7. $T_{pow} : T_k \,\hat{}\, T_m \rightarrow T_{\max(k,m)}$
8. $T_{>} : T > T \rightarrow B$
9. $T_{<} : T < T \rightarrow B$
10. $T_{\geq} : T \geq T \rightarrow B$
11. $T_{\leq} : T \leq T \rightarrow B$
12. $T_{=} : T = T \rightarrow B$

A generalized implementation of the arithmetic operators is done by realization of the interface called `IArithmeticOperatorsProvider` and its concrete implementation developed in the general delegate class `GenericArithmeticOperatorsDelegate`. This design and implementation allow not only further exposure of kinds-as-first-class values later on after several iterations of refinement, but also will allow operator and type overloading or replacement of type handling implementation altogether if some researchers wish to do so. The implementation of the engine, GEE, is thus changed, to only refer to the interface type implementation when dealing with these operators. Equivalently for the logic comparison operators we have the `ILogicComparisonOperatorsProvider` class and the corresponding `GenericLogicComparisonOperatorsDelegate` class. The latter relies on the comparator implemented for the numerical kind, such as `NumericComparator`. Using comparators (i.e. classes that implement the standard `Comparator` interface) allows Java to use and to optimize build-in sorting and searching algorithms for collections of user-defined types. In our case, the class called `GenericLogicComparisonOperatorsDelegate` is the implementation of the delegate class that also relies on it. The example for the numeric types for the described design is in Figure 1.

It is important to mention, that grouping of the numeric kind of integers and floating-point numbers does not violate the IEEE 754 standard [5], as these kinds implementation-wise wrap the corresponding Java's types (which are also grouped under numeric kind) and their semantics including the implementation of IEEE 754 by Java in accordance with the Java Language Specification.

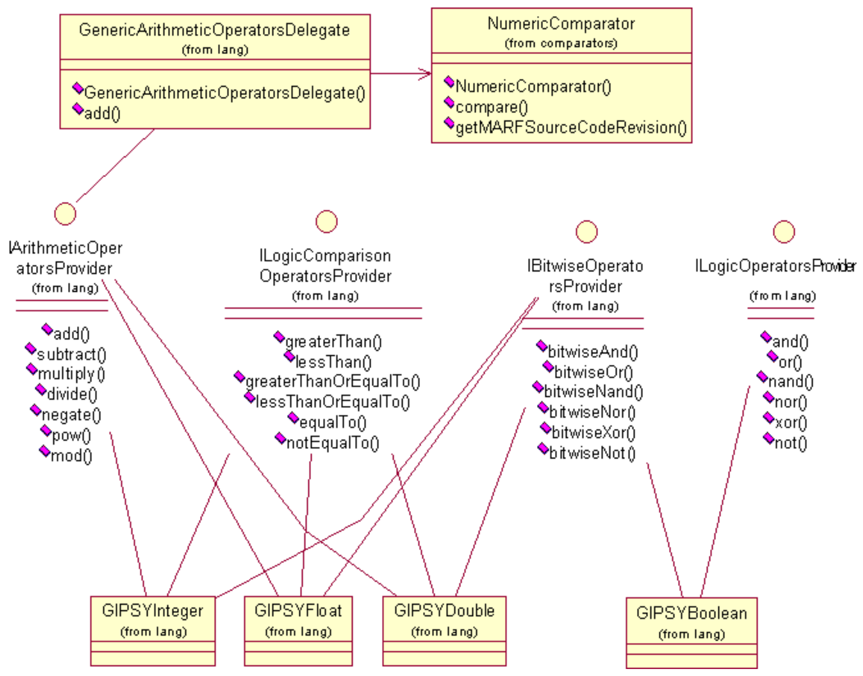

Figure 1: Example of Provider Interfaces and Comparators

2.1.2 Logic Kind

Similarly to numeric types, the primitive type `GIPSYBoolean` fits the category of the types that can be used in the Boolean expressions. The operations the type provides expect the arguments to be of the same type – Boolean. The following set of operators on the logic type B we provide in the GIPSY type system:

1. $B_{and} : B \bigwedge B \rightarrow B$
2. $B_{or} : B \bigvee B \rightarrow B$
3. $B_{not} : \neg B \rightarrow B$
4. $B_{xor} : B \bigoplus B \rightarrow B$
5. $B_{nand} : \neg(B \bigwedge B) \rightarrow B$
6. $B_{nor} : \neg(B \bigvee B) \rightarrow B$

Note that the logical XOR operator (denoted as $\bigoplus$) is different from the corresponding bitwise operator in Section 2.1.3 similarly to as the bitwise vs. logical are for AND and OR. Again, similarly to the generalized implementation of arithmetic operators, logic operator providers implement the `ILogicOperatorsProvider` interface, with the most general implementation of it in the delegate class `GenericLogicOperatorsDelegate`.

2.1.3 Bitwise Kind

Bitwise kind of types covers all the types that can support bitwise operations on the the entire bit length of a particular type T. Types in this category include the numerical and logic kinds described earlier in

Section 2.1.2 and Section 2.1.1. The parameters on both sides of the operators and the resulting type are always the same. There are no implicit compatible type casts performed unlike for the numeric kind.

1. $T_{bit-and} : T \& T \rightarrow T$
2. $T_{bit-or} : T | T \rightarrow T$
3. $T_{bit-not} : !T \rightarrow T$
4. $T_{bit-xor} : T \bigotimes T \rightarrow T$
5. $T_{bit-nand} : !(T \& T) \rightarrow T$
6. $T_{bit-nor} : !(T | T) \rightarrow T$

The generalized implementation of this kind's operators is done through the interface that we called `IBitwiseOperatorsProvider`, which is generically implemented in the corresponding delegate class `GenericBitwiseOperatorsDelegate`.

2.1.4 Composite Kind

As the name suggests, the composite kind types consist of compositions of other types, possibly basic types. The examples of this kind are arrays, structs, and abstract data types and their realization such as objects and collections. In the studied type system these are `GIPSYObject`, `GIPSYArray`, and `GIPSYEmbed`. This kind is characterized by the constructors, dot operator to decide membership as well as to invoke member methods and define equality. The design of of these types, just like the entire type system, adheres to the Composite design pattern. The most prominent examples of this kind are in Figure 2, including `GIPSYContext` which is composed of `Dimensions` and indirectly of the `TagSets`.

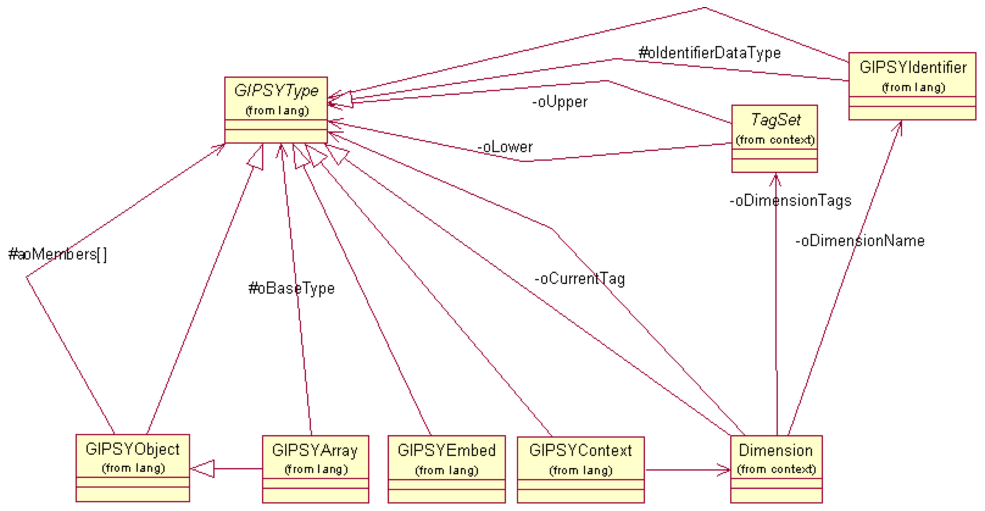

Figure 2: Composite Types.

2.1.5 Intensional Kind

The intentional types kind primarily has to do with encapsulation dimensionality, context information and their operators. These are represented by the types `GIPSYContext`, `Dimension`, and `TagSet`[1] types. The

[1]Not mentioned here; please refer to [19, 15].

common operators on these types include the context switching and querying operators @ and # as well as context calculus operators. Additional operators can be included depending on the intensional dialect used, but the mentioned operators are said to be the baseline operators that any intensional language can be translated to use. Implementation-wise there is a `IContextSetOperatorsProvider` interface and its general implementation in `GenericContextSetOperatorsDelegate`. The context calculus operators on simple contexts include standard set operators, such as **union**, **difference**, **intersection**, and Lucx-specific **isSubContext**, **projection**, **hiding**, and **override** [19, 15].

2.1.6 Function Kind

The function types represent either "functional" functions, imperative procedures, and binary and unary operators that, themselves, can be passed as parameters or returned as results. In our type system these are represented by `GIPSYFunction`, `GIPSYOperator`, and `GIPSYEmbed`. The common operators on this include equality and evaluation.

3 Describing Some GIPSY Types' Properties

To show the overview of the properties of GIPSY types to get a better and more complete understanding of the spectrum of their behavior we cover them in light of description and comparison to existential, union, intersection, and linear types.

3.1 Existential Types

All of the presented GIPSY types are existential types because they represent and encapsulate modules and abstract data types and separate their implementation from the public interface specified by the abstract `GIPSYType`. These are defined as follows:

$T = \exists \texttt{GIPSYType}\{\texttt{Object}\ a;\ \texttt{Object}\ \texttt{getEnclosedTypeObject}();\ \texttt{Object}\ \texttt{getValue}();\}$

The above is implemented in the following manner, e.g.:

$boolT\ \{\texttt{Boolean}\ a; \texttt{Object}\ \texttt{getEnclosedTypeObject}();\ \texttt{Boolean}\ \texttt{getValue}();\}$
$intT\ \{\texttt{Long}\ a; \texttt{Object}\ \texttt{getEnclosedTypeObject}();\ \texttt{Long}\ \texttt{getValue}();\}$
$doubleT\ \{\texttt{Double}\ a; \texttt{Object}\ \texttt{getEnclosedTypeObject}();\ \texttt{Double}\ \texttt{getValue}();\}$
... and so on.

All these correspond to be subtypes of the the more abstract and general existential type T. Assuming the value $t \in T$, then $t.\texttt{getEnclosedTypeObject}()$ and $t.\texttt{getValue}()$ are well typed regardless the what the `GIPSYType` may be.

3.2 Union Types

A union of two types produces another type with the valid range of values that is valid in *either* of the two; however, the operators defined on the union types must be those that are valid for *both* of the types to remain type-safe. A classical example of that is in C or C++ the range for the `signed char` is $-128\ldots127$ and the range of the `unsigned char` is $0\ldots255$, thus:

$\texttt{signed char} \cup \texttt{unsigned char} = -128\ldots255$

In C and C++ there is a `union` type that roughly corresponds to this notion, but it does not enforce the operations that are possible on the union type that must be possible in both left and right types of the uniting operator. In the class hierarchy, such as in GIPSY, the union type among the type and its parent is the parent class; thus, in our specific type system the following holds:

$\forall T \in G : T \cup \texttt{GIPSYType} = \texttt{GIPSYType}$
$\texttt{GIPSYArray} \cup \texttt{GIPSYObject} = \texttt{GIPSYObject}$
$\texttt{GIPSYVoid} \cup \texttt{GIPSYBoolean} = \texttt{GIPSYBoolean}$
$\texttt{GIPSYOperator} \cup \texttt{GIPSYFunction} = \texttt{GIPSYFunction}$

where T is any concrete GIPSY type and G is a collection of types in the GIPSY type system we are describing. Equivalently, the union of the two sibling types is their common parent class in the inheritance hierarchy. Interestingly enough, while we do not explicitly expose kinds of types, we still are able to have the following union type relationships defined based on the kind of operators they provide as siblings under a common interface:

$\forall T \in A : T \cup \texttt{IArithmeticOperatorProvider} = \texttt{IArithmeticOperatorProvider}$
$\forall T \in L : T \cup \texttt{ILogicOperatorProvider} = \texttt{ILogicOperatorProvider}$
$\forall T \in B : T \cup \texttt{IBitwiseOperatorProvider} = \texttt{IBitwiseOperatorProvider}$
$\forall T \in C : T \cup \texttt{IContextOperatorProvider} = \texttt{IContextOperatorProvider}$
$\forall T \in D : T \cup \texttt{ICompositeOperatorProvider} = \texttt{ICompositeOperatorProvider}$
$\forall T \in F : T \cup \texttt{IFunctionOperatorProvider} = \texttt{IFunctionOperatorProvider}$

where T is any concrete GIPSY type and A is a collection of types that provide arithmetic operators, L – logic operators providers, B – bitwise operators providers, C – context operators providers, D – composite operator providers, and F – function operator providers. Thus:

$\{\texttt{GIPSYInteger}, \texttt{GIPSYFloat}, \texttt{GIPSYDouble}\} \in A$
$\{\texttt{GIPSYBoolean}\} \in L$
$\{\texttt{GIPSYInteger}, \texttt{GIPSYFloat}, \texttt{GIPSYDouble}, \texttt{GIPSYBoolean}\} \in B$
$\{\texttt{GIPSYContext}, \texttt{Dimension}\} \in C$
$\{\texttt{GIPSYObject}, \texttt{GIPSYArray}, \texttt{GIPSYEmbed}\}, \texttt{GIPSYString}\} \in D$
$\{\texttt{GIPSYFunction}, \texttt{GIPSYOperator}, \texttt{GIPSYEmbed}\} \in F$

Another particularity of the GIPSY type system is the union of the string and integer types under dimension:

$\texttt{GIPSYInteger} \cup \texttt{GIPSYString} = \texttt{Dimension}$

and this is because in our dimension tag values we allow them to be either integers or strings. While not a very common union in the majority of type system, they do share a common set of tag set operators defined in [20] for ordered finite tag sets (e.g. `next()`, etc.).

3.3 Intersection Types

Intersection type of given two types is a range where the sets of valid values overlap. Such types are safe to pass to methods and functions that expect either of the types as the intersection types are more restrictive and compatible in both original types. A classical example of an intersection type if it were implemented in C or C++ would be:

$\texttt{signed char} \cap \texttt{unsigned char} = 0 \ldots 127$

The intersection types are also useful in describing the overloaded functions. Sometimes the are called as refinement types. In the class hierarchy, the intersection between the parent and child classes is the most derived type, and the intersection of the sibling classes is empty. While the functionality offered by the intersection types is promising, it is not currently explicitly or implicitly considered in the GIPSY type system, but planned for the future work.

3.4 Linear Types

Linear (or “uniqueness”) types are based on linear logic. The main idea of these types is that values assigned to them have one and only one reference to them throughout. These types are useful to describe immutable values like strings or hybrid intensional-imperative objects (see [24] for details). These are useful because most operations on such an object “destroys” it and creates a similar object with the new values, and, therefore, can be optimized in the implementation for the in-place mutation. Implicit examples of such types in the GIPSY type system are `GIPSYString` that internally relies on Java’s `StringBuffer` that does something very similar as well as the immutable `GIPSYObject` is in JOOIP [24] and immutable `GIPSYFunction`. Since we either copy or retain the values in the warehouse, and, therefore, one does not violate referential transparency or create side effects in, and at the same time be more efficient as there is no need to worry about synchronization overhead.

4 Conclusion

Through a series of discussions, specification, design, and implementation details we presented a type system used by the GIPSY for static and dynamic type checking and evaluation of intensional and hybrid intensional-imperative languages. We highlighted the particularities of the system that does not attribute to a particular specific language as traditionally most type systems do, but to an entire set of languages and hybrid paradigms that are linked through the type system. We pointed out some of the limitations of the type system and the projected work to remedy those limitations. Overall, this is a necessary contribution to GIPSY-like systems to have a homogeneous environment to statically and dynamically type-check intensional and hybrid programs.

4.1 Future Work

The type system described in this paper has been implemented in the GIPSY. However, due to recent changes including the introduction of contexts as first-class values in *Generic Lucid*, as well as the development of the fully-integrated hybrid language JOOIP [24], our implementation still needs thorough testing using complex program examples testing the limits of the type system. Additional endeavours, noted in the previous sections of this paper, include:

- Exposing more operators for composite types to Lucid code segments.
- Allowing custom user-defined types and extension of the existing operators and operator overloading.
- Expose *kinds* as first class entities, allowing programs more explicit manipulation of types.
- Consider adding intersection types for more flexibility in the future development of the type system, allowing more type casting possibilities at the programming level.

4.2 Acknowledgments

This work was funded by NSERC and the Faculty of Computer Science and Engineering, Concordia University.

References

[1] E. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge. *Multidimensional, Declarative Programming*. Oxford University Press, London, 1995.

[2] E. A. Ashcroft and W. W. Wadge. Erratum: Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 6((1):200), 1977.

[3] P. Grogono. GIPC increments. Technical report, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Apr. 2002.

[4] P. Grogono, S. Mokhov, and J. Paquet. Towards JLucid, Lucid with embedded Java functions in the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 15–21. CSREA Press, June 2005.

[5] IEEE. 754-2008: IEEE standard for floating-point arithmetic. [online], Aug. 2008. `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[6] B. Lu. *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2004.

[7] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall, 4 edition, 1997.

[8] S. Mokhov and J. Paquet. General imperative compiler framework within the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 36–42. CSREA Press, June 2005.

[9] S. Mokhov and J. Paquet. Objective Lucid – first step in object-oriented intensional programming in the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 22–28. CSREA Press, June 2005.

[10] S. A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005. ISBN 0494102934.

[11] C. B. Ostrum. *The Luthid 1.0 Manual*. Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[12] J. Paquet. *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[13] J. Paquet, P. Grogono, and A. H. Wu. Towards a framework for the general intensional programming compiler in the GIPSY. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, Canada, Oct. 2004. ACM.

[14] J. Paquet and P. Kropf. The GIPSY architecture. In *Proceedings of Distributed Computing on the Web*, Quebec City, Canada, 2000.

[15] J. Paquet, S. A. Mokhov, and X. Tong. Design and implementation of context calculus in the GIPSY environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[16] J. Paquet and A. H. Wu. GIPSY – a platform for the investigation on intensional programming languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 8–14, Las Vegas, USA, June 2005. CSREA Press.

[17] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge. Sequential demand-driven evaluation of eager TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1266–1271, Turku, Finland, July 2008. IEEE Computer Society.

[18] T. Rahilly and J. Plaice. A multithreaded implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1272–1277, Turku, Finland, July 2008. IEEE Computer Society.

[19] X. Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Apr. 2008.

[20] X. Tong, J. Paquet, and S. A. Mokhov. Context Calculus in the GIPSY. Unpublished, 2007.

[21] E. Vassev and J. Paquet. A generic framework for migrating demands in the GIPSY's demand-driven execu-

tion engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 29–35, Las Vegas, USA, June 2005. CSREA Press.

[22] W. Wadge and E. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[23] K. Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[24] A. Wu, J. Paquet, and S. A. Mokhov. Object-Oriented Intensional Programming: A New Concept in Object-Oriented and Intensional Programming Domains. Unpublished, 2007.

[25] A. H. Wu and J. Paquet. Object-oriented intensional programming in the GIPSY: Preliminary investigations. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 43–47, Las Vegas, USA, June 2005. CSREA Press.